January 2010

Intelligent Speaker Design

James Corell
Worcester Polytechnic Institute

WORCESTER POLYTECHNIC INSTITUTE

Intelligent Speaker Design

A Java IQP in Acoustics

James Corell

Fall 2009

Contents

Introduction

As the twenty-first century begins, the age of technology continues to press forward, and the digital integration of even the simplest of tasks becomes more and more prevalent. The dawn of computers, which once were strictly contained to research labs with vast rooms and extensive funding, has had an enormous societal impact on the way humans conduct their daily lives, and this influence continues to grow as newer and smaller integrated computing becomes available. As the availability of such computing power continually increases, so too the possibilities for the use of such technology expand past boundaries of what was previously thought possible. The developments in the fields of processing and wireless networks especially have created an era of communication that is unparalleled in its functionality and efficiency. Data can be published from computers and accessed in real-time by other computers, no matter where across the globe they might be. This continual exchange of information opens up endless possibilities of communication functionality and global interactivity.

With this constant flow of data comes a new age of large-scale computer-based sensor systems, where information from around the world is constantly broadcasted and interpreted. Hundreds of machines placed around a room, across a city, or even on different continents can share the information they have gleaned, whether it be temperature data, musical notes, or whatever else the end-user desires to relay. In addition to the sharing of stored sensor data, the potential for control systems on a global scale is enormous. Consider the application of simultaneous control over any number of global sites, each containing sensor-equipped computers. These machines can now be remotely managed in order to synchronize whatever their pre-programmed actions or data responses might be. This real-time harmonization of large-scale systems carries incredible implications for the future of interactive networking applications.

Project Definition

Consider now the traditional audio reproduction system, and visualize how this technology could be used to fill the halls of the future with sound. Imagine the rooms of a museum, each of them filled with musical tones, and every one of them linked together to form a responsive network. The speakers themselves become more than simply circuits employed to move a speaker coil; instead they become a source of processing and data input themselves, and they gain a level of intelligence previously unheard of. As a networked computer is added internally to the speaker itself, every speaker placed around the room has the potential to read in sensory information about the room it is placed and react accordingly. As a room fills with people, and the temperature rises, the sensory data changes, and now the ambient atmosphere can now be adjusted in response. Algorithms may be developed to continually adapt certain elements of the aural presentation in order to invoke a desired feeling in the listeners. If an additional stand-alone computer is assigned to the room, now these speakers can be monitored over the network via a server application. This opens more doors for further development of the system, as this enables not only the constant monitoring of sensor data, but now advanced synchronization becomes possible. The end-user can use advanced algorithms to link the behavior of all speakers involved in order to manipulate the ambient audio atmosphere to an even further degree.

Project Abstract

The overarching idea behind this project is the creation of nodes of intelligent computer-driven speaker cabinets that can use sensors and algorithms to provide an interactive and varying aural atmosphere. These devices would have the capability of passing messages to other nodes, both within their own environment and around the world. Sensory data and processing would enable the creation of a subtle audio environment tailored to the mood of the location, the number of people in a room, or any other arbitrary data fed into the system. This enterprise is of a rather large nature however, so the specific focus was on the creation of the software framework necessary for the more abstract concepts to be implemented.

Background

Upon conducting significant research into the types of devices under consideration, no system was found that resembled the desired end-product. A number of assorted U.S. patents have been issued for related technology, notably several forms of network-driven speaker cabinets, but none to the extent this project desired to carry the ideas.

- U.S. patent 6766025 describes a speaker designed to self-compensate for environmental distortion and equalization of the output signal. The device is equipped with a microphone and a DSP section. A reference signal is sent to the speaker, the speaker output is compared with the original signal, and the resulting difference is used by the DSP section to calculate any necessary adjustments. While this example exemplifies the use of processing internal to the cabinet itself, there is no networking functionality.

- U.S. patent 7031476 was filed for a speaker design with a built-in communication system. The purpose was however limited strictly to maintenance communication

with the amplifier, which was used ensure that amplifier was configured correctly for the characteristics of the speaker, there was no internal processing.

- U.S. patent 6329908 was filed for the design of an addressable speaker system, or fundamentally a set of connected speakers. It was designed to work with an RF location system, which would provide the location of personnel wearing a certain badge. Upon receipt of the location, the audio network would find the speaker closest to the individual wearing the badge, and a central processor determines through which speaker to play a pre-determined audio signal. This exemplifies the idea of a network of speakers, but it contains no internal network computing or processing to adjust the speaker output in any way.

- U.S. patent 7197148 describes a network of speakers whose outputs are controlled separately from a central remote processing system. This system can selectively send commands and change output characteristics to whichever speakers the user desires. This system resembles the configuration in question for this project but however lacks individual computing power internal to the devices, all signals are centrally generated.

- http://www.prevelakis.net/Papers/inc2004.html. Lastly, several students from Drexel University wrote about a similar concept in 2004. They envisioned a computerized audio reproduction system where audio signals are streamed to each speaker over Ethernet. Their ideas included network functionality of a type similar to what we desired, but they conceived a network exclusively for the purpose of streaming audio, not for any further communicative uses.

Speaker Design Requirements

The requirements for the design of the speakers themselves are quite straightforward. The idea is that a small computer, most likely a Mac Mini, would be placed inside a small tube-like enclosure. This enclosure would have external speakers, which would be fed from the audio output of the computer resting inside. The only essential hardware requirements are stable operation, audio storage and output capability, and wireless connectivity. Further demands may become necessary for future implementations, but to accomplish the proof of functionality for this project and to demonstrate the abstract concepts, the basic hardware itself is very straightforward.

Software Requirements

The specific goal of the project was to develop a customized network architecture for wireless communication between groups of speakers, called nodes. The goal of this architecture was several-fold. The first and main objective was to facilitate simple communication between these nodes in whatever fashion is most appropriate and efficient. One node must be able to take a message from another node or from a central server and execute the proper command. This functionality provides a whole additional layer of interactivity in these speaker designs. Secondly, if a central server is desired, the connectivity provides an easy method of monitoring the status of a node of speakers. A central server could keep track of what different nodes are doing, or a server inside the node can monitor the status of each speaker enclosure itself. The integration of communication and monitoring brings into play a whole field of abstract creativity and art, the bounds of which are limitless. For example, the nodes may monitor how many people are in the room and adjust some aspect of the audio environment accordingly. It was crucial that the design of such a network be exceedingly thorough and thought-out, as the actual implementation of the system is to be accomplished by other student(s) in the future. It was imperative that a meticulous set of both features and design requirements be created in anticipation of what will be achieved at a later point in time.

Design Requirements

- Adaptability. The idea of this project itself is very much an abstract one; the final functionality has been by no means set in stone. Once the core networking capability is in place inside these speaker cabinets, the possibilities of what can be done with this system are absolutely endless. The number of computers involved is itself subject to a great degree of change, and the set of features, especially those of a more experimental or artistic nature, are far from being decided upon. So, the design of the network architecture itself must be very deliberate and thought-out. If no planning or structure is put into the organization of the written code, the main idea behind the undertaking will be nullified. The network itself is not a final goal; it is simply a tool used to support the almost metaphysical and theoretical concepts driving the underlying project. Therefore, adaptability is the single most important property of this compilation of written code.

- Simple Architecture. The time constraint for the project and desire for adaptability demanded a simple network structure unhindered by fancy coding or application-specific features. The desire was for a simple communications system upon which any future developments could be written.

- Network Stability. Given that the planned distribution of this device is on a global scale, low maintenance was an important feature. The network functionality of the system had to be made stable as possible, for constant maintenance or a large IT staff are not feasible.

- Low Bandwidth Usage. The nature of these devices required that they be functional wherever around the globe they may happen to be placed. Therefore, given our lack of knowledge regarding the speed of the network connections to be used, it was imperative that the amount of network traffic be kept on a fairly small scale. Sending and receiving constant

streams of packets could potentially be the downfall of the system if the local network in question was not sufficiently capable.

- Efficient Networking Structure. The potentially global scale of this project demanded a certain level of organization in the structure of the network traffic. If every machine in existence could instantly send its sensor data anywhere, the lack of an organized data routing scheme would cause the system to be crippled from inefficiency. The sensible solution then was some form of central traffic management system, which would also more easily facilitate central monitoring of speakers placed around the world.

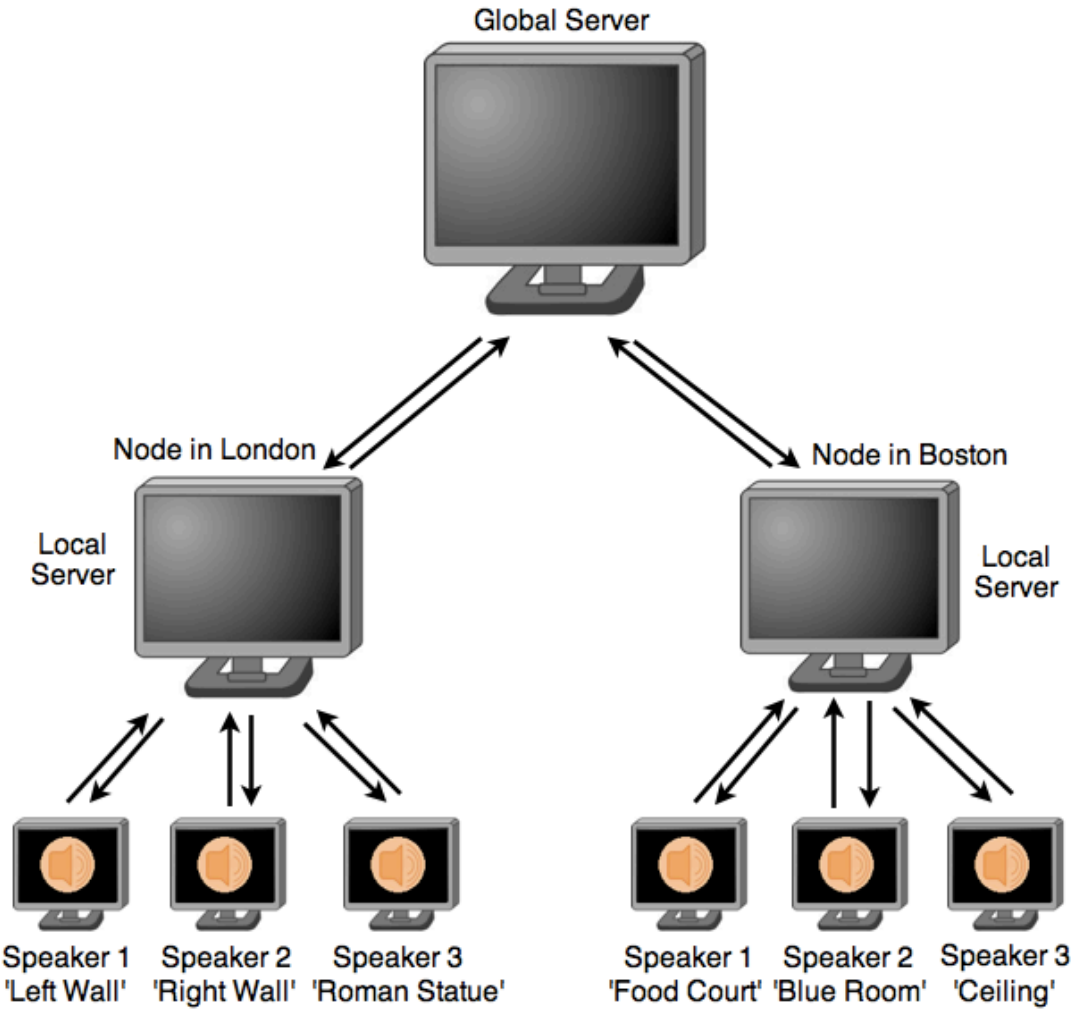
Design Strategy and Choices

With these requirements in mind, the decision was made to write the software in Java. The flexibility it offered was tremendous, and the language itself is very high level when compared to the other choices offered. This made the coding of the network communication system much simpler, as everything needed on a lower system level to make the physical connections happen is taken care of by Java itself. The other main advantage of Java is its cross-compatibility. Although the desire is to embed a Mac Mini inside of each speaker, the ability to launch the applications on any computer, regardless of operating system, make or model, is a specification very much in line with our paramount goal of adaptability.

Network Design

It became apparent that it would simply not be enough for all the computers in use to communicate with each other on the same level. It was evident that a central server would become necessary to aid in communications between the nodes on a global level. This server would not only be able to monitor nodes placed around the world, but should a node desire to share its information with another node, this server would be able process the data and redirect it to its intended destination. This relaying functionality would keep the network sensibly organized and routed properly. The machine could be placed anywhere, as its physical location would be completely irrelevant. This kind of setup provided a certain level of efficiency and straight-forwardness to messages transmission, as well as the ability to very simply monitor every node and get frequent status updates. However, the traffic that was generated to the central server solely from regular status updates was still more frequent than desired. The resolution was to place an intermediate server at each site, or node. This server would handle basic communication and updates within the context of any local speakers, and the global server would receive only basic status updates from the node as a whole.

Figure 1: Network Structure Diagram



This diagram shows an example of the organizational hierarchy of the network, and the arrows show the flow of any communication or exchange of data that might occur. Should the local server in London desire to share data with the local server in New York, that data will be passed to the global server with instructions to redirect the data to its intended recipient. Every message or command gets sent up the chain first and then re-sent if needed. At no point can any machine speak directly to another machine on the same network level; this eliminates the need for each computer in use to open a connection with every other computer, and the network traffic becomes completely centralized. Three different software applications were written; each one for each level of machine in the network.

The Three Components

Speakers:

The speakers are designed to be run from a small computer placed inside the speaker enclosure. This computer will run the speaker version of the software, which has basic communication functionality with any other local speakers. This is accomplished through the local server via wireless Ethernet. Sensor hardware of any type could be attached to obtain data from wherever in the room the device was placed.

Local Server:

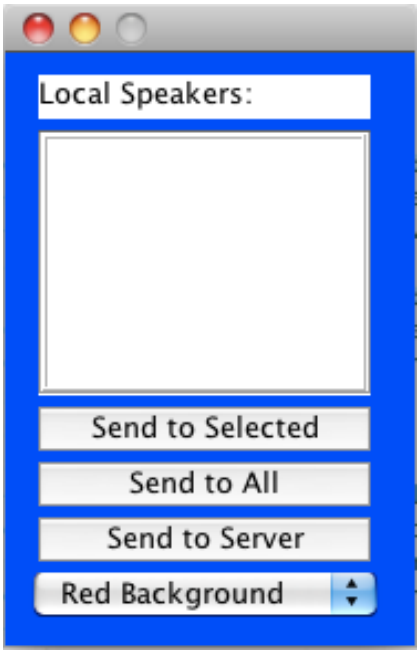
This computer acts as a server to all the local speakers, for which it monitors status updates and redirects commands inside the network as needed. It would not contain any sensor hardware most likely, but it would have the capability to process incoming data and send universal commands. Its instant connection to everything on the network enables complete audio synchronization, whether it be matching volume levels, tracking the movement of people, or applying some developmental algorithm to achieve a psychological effect.

Global Server:

This server behaves very similarly to the local servers, but instead of managing speakers, it manages the local servers, relays them instructions, and keeps them in synchronization. This software maintains a global view of the system as a whole. Any and all machines of any type can be monitored and controlled from where this central system is placed.

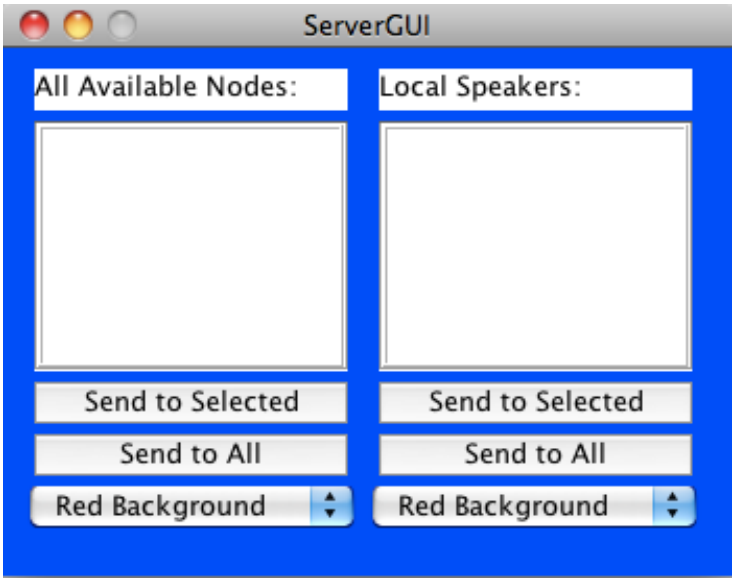
Software Description

Speakers:



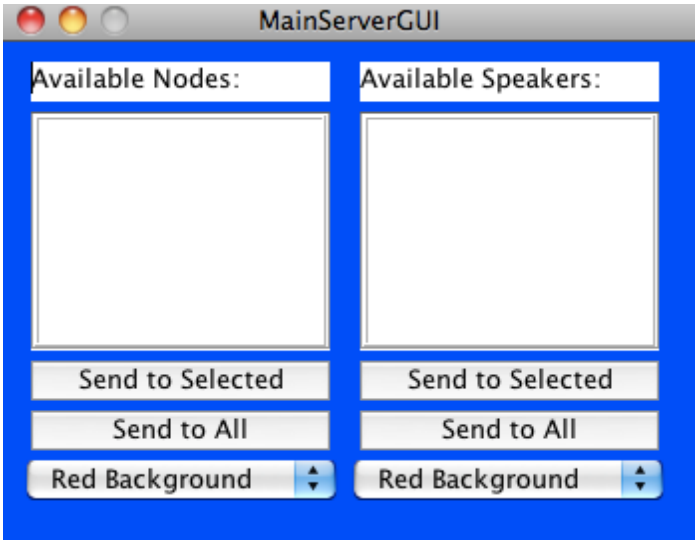
The interface designed for the speaker software was created to demonstrate the concepts of the desired functionality. The box at the top of the screen displays the list of all available speakers in the local node. To keep this list up to date, the local server sends updates to all its clients when the list changes. The buttons provide the ability to send commands to other machines on the network. The “Send to Selected” button sends a command solely to whatever speaker in the list is selected, while “Send to All” sends the same command to all speakers. The “Send to Server” button provides communication directly with the local sever. Each of those buttons is used to send whichever command is currently selected in the drop-down box, which provides a short list of sample commands used to demonstrate the software.

Local Server:



The local server software has a slightly more expanded interface. The right-hand list contains the names of any speakers that are connected to the local server, which is automatically updated. When a machine connects to the server, it is added to the list, and it sends regular status updates to indicate whether it is still functional. These status updates are reflected in the color of the entry in the list: green implies that all is well, while red means the server has not received a recent update. The left-hand list contains the names of all available nodes world-wide, and the color also reflects the status of nodes. This list is automatically updated from the global server every time it changes. The buttons and dialog box on each side perform the same function for different targets. Similar to the GUI for the speaker, they are used to send commands or messages to other applications in existence on the network, the left side to local servers in other nodes, and the right side to any or all connected speakers. The dialog box contains the names of sample commands to demonstrate, while the “Send to Selected” and “Send to All” buttons let the user decide to whom the selected command will be sent.

Global Server:



The global server's application is almost identical to that of the local server. The buttons and boxes perform the same function as those of the local server, but the scrolling lists behave differently. The left-hand list still contains the names of all available nodes, but it is here that the list is generated. Each local server connects to the global server, and upon successful connection, it is shown in the list. Its color again signifies its status; green means all is well, while red indicates the lack of recent data traffic. The right-hand list labeled "Available Speakers" contains the list of available speakers connected to whichever node is selected. When a node is selected in the left-hand list, a command is sent the node. The node's local server then returns the names of all its connected speakers, and these names are displayed in the right-hand list. The local server continues to update these names while the same node is still selected in the global server application. Thus the global server has ultimate control over all machines on the network worldwide. It not only can send commands to any and all nodes selectively, but as each node is highlighted, the associated speakers can be controlled individually or as a group.

Technical Description

This section will attempt to describe the technical functionality of the code written for this project, in the hopes that someone will be able to take the applications and use them in some experimental acoustical system. The code itself is thoroughly documented, so what will be presented here will not be a complete explanation of the code, but instead it will be a outline of the technical concepts used design the system. The content will assume the reader has a basic understanding of both general programming practices and Java.

The Core Network Communication Infrastructure:

Java's high level architecture means the programmer can disregard the low-level mechanics of the physical connections. The basic structure of a connection uses a server/client organization, which is accomplished very simply in Java. The command:

```
ServerSocket server_name=new ServerSocket(4444);
```

creates a new ServerSocket using the port 4444. This object will listen for incoming connections on port 4444. Upon the creation of a client pointed to the server IP address and port number,

```
Socket socket_name=new Socket(192.168.0.4, 4444);
```

the server accepts the connection and stores its details in a socket:

```
Socket new_socket = server_name.accept();
```

Sending data over the network requires opening streams to send and receive. For every connection, both machines involved must open both an input and output stream, like so:

```
ObjectOutputStream out=new ObjectOutputStream(clientSocket.getOutputStream());

ObjectInputStream in=new ObjectInputStream(clientSocket.getInputStream());
```

As their names imply, these streams not only send strings of data but are also capable of sending an instance of any class that implements the Serializable interface. This enables a highly efficient

data transmission structure where several pieces of primitive data can be written to the stream simultaneously inside of an encompassing object.

The following two lines of codes write a piece of data to the output stream and then flush the stream. This ensures the message was sent and not left sitting in the buffer.

```
out.writeObject(object_name);

out.flush();
```

The receiving machine meanwhile loops through and continuously attempts to read input from the stream and process it. This function call does not continue forward until data is received.

```
Object input=in.readObject();
```

For all data transmission, a user-written wrapper class named Command is used. This class contains four pieces of data and several functions associated with them:

- *action* - a String containing the name of the command action.
- *data* - an Object containing any data associated with the action of the command.
- *redirect* - a boolean indicating if this message is intended for the sender or should be redirected.
- *recipient* - a String indicating what machine the message should be redirected to.

For example, for a machine to send its name to the parent server, the following code is executed, where the variable *name* contains the name of the machine.

```
output.writeObject(new Command(“Name”,name));
```

Here is how the local servers return their list of speakers to the global server:

```
output.writeObject(new Command("Speaker List",parent.getActiveSpeakers()));
```

Global Server:

As many other pieces of the system also exemplify, the global server extends the network communication ideas in order to host multiple clients. In order for a server to maintain connections with more than one client, the server uses multi-threaded networking. For every connection made, the server creates a thread to handle all input, output, and processing. All of these threads are stored inside a Hashtable, so they can be recalled later. A Hashtable enables the user to store data like an array, but every piece of data has a specific key to recall it instead of an index. Every thread is stored with the name of connected machine as the key. A separate class created to handle the business of the threads, and one instance of the class is created for each thread. This class that handles the threads, called MainServerThread, does a number of things:

- Instantiates the input/output streams and any other necessary objects.
- Starts the thread, which executes the run-time loop. This loop continuously waits for input, which arrives as Command objects. Once a command is received, a series of if statements read the name associated with the command and determine what action is appropriate to take.
- A thread-based sub-class, called MainServerTimingThread, is created to check for receipt of recent communication. This dedicated thread is required, for the method to read input in the main thread hangs upon each execution until it has received data, so any processing that is dependent on regular time intervals must be done separately.

These two threads handle all monitoring for and responding to input over the network. To handle user input, i.e. clicking a button, the standard ActionListener style code is used. This creates code in a particular class which will be called instantly when each button is pressed. The main application is added as the ActionListener reference for all buttons except the “Send to All” button on the list of active nodes. For this button, each instance of MainServerThread is added as the ActionListener reference as it connects to the global server. Upon the generation of the button-event, the Java virtual machine automatically calls upon each of the handling threads to respond.

Local Server:

The technical structure of the local server software is very similar to that of the local server. A thread class is instantiated to handle each connection and its related input gathering, processing, and status updates. The main application stores each of these threads in a Hashtable with the name of the speaker as the key. The class ChildServerThread contains the individual threading code and has virtually the same structure as that of MainServerThread:

- Instantiate the input/output streams and any other necessary objects.
- Start the thread, which executes the run-time loop. This loop continuously waits for input, which arrive as Command objects. Once a command is received, a series of if statements read the name associated with the command and determine what action is appropriate to take.
- A second thread class, called ChildServerTimingThread, is created to check for receipt of recent communication. This dedicated thread is required, for the method to read input in the main thread hangs upon each execution until it has received data, so any processing dependent on regular intervals of time must be done separately.

Here the multi-threading structure differs from that of the global server. The concept of multi-threading is that every running application must establish a thread to handle each connection. In our case two threads are established for each connection, one for input processing, and one for regular status updates, which function on regular timing intervals. So in addition to a pair of threads for each communication channel with a speaker, a pair of threads is also needed to maintain the connection to the global server. Similar to the child threads, a separate class was created to handle the threading of this connection. The design of this class, called ParentServerThread, is identical to the other thread-based classes. One thread is created to handle the constant processing of input, while another is created to run the timer that generates regular status updates.

Speakers:

The framework behind the speaker software is naturally of a simpler nature. The application run by each speaker only maintains a single two-way connection with the local server, so unlike the other applications, no multi-threading is not necessary. After initializing all objects associated with the window, the program runs through a loop to check for user input. This loop waits for network traffic, in the form of commands sent from elsewhere on the network, and then responds accordingly. The ActionListener interface utilized on all user-interaction objects responds to all user input, while a single free-running thread, instantiated as the subclass ClientTimingThread, services the transmission of regular updates to the local server.

Possible Usage

The goal was to create a system that could be employed for whatever end product the user might desire to deploy. The unique ability to both generate synchronization commands on a global level and to gather individual sensor readings on a local level creates boundless possibilities for future usage The beauty of the code written was in that no final implementation was chosen or developed. Everything created was designed for a bare-bones communications framework that would be as generic as possible. A few exampled applications were however initially envisioned to keep the project on the right track.

- A simple temperature monitoring system could be placed in each speaker. As people filled different parts of the room, the temperature would rise from the number of bodies in the room, and the volume in each section of the room could be automatically adjusted to compensate for the increased background noise.

- A number of speakers could be placed to sonically fill a large room pleasantly, possibly a museum exhibit or information display. The local servers would contain complex time-variant algorithms, either user-generated or garnered in real-time from a website, such as the geomagnetic data from the Earth's magnetic field. This information would be passed on in chunks to the speakers over the network. The algorithms would be chosen for their similarity to certain naturally resonant frequencies in the brain. The algorithmic data would then be used to adjust the volume or some tonal characteristic of whatever audio was being produced in such a way as to achieve some psychologically pleasing effect.

Problems

While the system is quite stable, it still produces the occasional runtime exception, but they are of a random and unpredictable nature. The occurrences are very few and far between and have not been successfully traced to any coding issues. Of these rare occurrences, the majority do not cause the application to terminate its run-time activities.

Java's networking packages also lacks proper low-level monitoring of network connections, so while the servers manually check for recent receipt of a status update, there is no native method to detect the closure of a network socket. So while a message can be sent to inform the server that a speaker is going disconnect, the servers cannot instinctively detect this of their own doing. Expanding this functionality proved to be far too complicated to be developed during the course of this project.

Future Considerations

The software presents a kind of "proof of concept" to demonstrate the desired functionality, and as such, only several commands are preprogrammed. The structure of the code was written with utmost flexibility, and more commands can be added with great ease. This requires two simple steps:

1. Edit the function *populateCommandList()* in the application sending the command. First add the name of the command and any data to the Hashtable of commands using this format:

   ```
   commandsTable.put("Yellow Background", Color.yellow);
   ```

   Also add the command action to a String array containing the names of all commands:

   ```
   commands=new String[]{"Play Sound","Red Background","Yellow Background"};
   ```

2. Now create an additional case statement in the recipient application. This statement checks for the name of the command and executes any code needed using the appropriate form:

```
else if(input.getAction().equals("Play Sound"))
{
   playAudioFile();
}
```

Conclusions:

The goal presented was to create a custom framework for Java-based network communication. As the final usage was not yet determined, versatility was paramount. The final code presented for the project successfully meets the objectives initially established. The three level networking provides a high level of efficiency for data transmission, while the structure of the generic communication classes provide opportunities for future expansion. The applications have undergone testing to ensure the proper behavior, and while minor bugs still exist, everything is functional as desired. The code is well-documented to allow for maximum adaptability by future users other than the author. Hopefully in the future the work accomplished here can be used in exciting new aural experiments for the advancement of intelligent acoustics.